Recap: Can a process overwrite kernel memory?
Privilege levels

- Each segment has a privilege level
  - DPL (descriptor privilege level)
  - 4 privilege levels ranging 0-3
Privilege levels

• Currently running code also has privilege level
  • “Current privilege level” (CPL): 0-3
  • Can access only less privileged segments
    – E.g., 0 can access 1, 2, 3
• Some instructions are “privileged”
  • Can only be invoked at CPL = 0
  • Examples:
    – Load GDT
    – MOV <control register>
      • E.g. reload a page table by changing CR3
Real world

- Only two privilege levels are used in modern OSes:
  - OS kernel runs at 0
  - User code runs at 3
- This is called “flat” segment model
  - Segments for both 0 and 3 cover entire address space
- But then... how the kernel is protected?
  - Page tables
Page table: user bit

- Each entry (both Level 1 and Level 2) has a bit
  - If set, code at privilege level 3 can access
  - If not, only levels 0-2 can access
- Note, only 2 levels, not 4 like with segments
- All kernel code is mapped with the user bit clear
  - This protects user-level code from accessing the kernel
TLB

- CPU caches results of page table walks
  - In translation lookaside buffer (TLB)
- Walking page table is slow
  - Each memory access is 200-300 cycles on modern hardware
  - L3 cache access is 70 cycles
TLB

- TLB is a cache (in CPU)
  - It is not coherent with memory
  - If page table entry is changes, TLB remains the same and is out of sync

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf0231000</td>
<td>0x1000</td>
</tr>
<tr>
<td>0x00b31000</td>
<td>0x1f000</td>
</tr>
<tr>
<td>0xb0002000</td>
<td>0xc1000</td>
</tr>
</tbody>
</table>

Same Virt Addr. No Change!!!
Invalidating TLB

- After every page table update, OS needs to manually invalidate cached values
- Modern CPUs have “tagged TLBs”,
  - Each TLB entry has a “tag” – identifier of a process
  - No need to flush TLBs on context switch
- On Intel this mechanism is called
  - Process-Context Identifiers (PCIDs)
Creating Processes
Recap: kernel memory layout
Process memory layout
How does kernel creates new processes?
How does kernel creates new processes?

- Exec
  - `exec("/bin/ls", argv);`
exec(): high-level outline

- We want to create the following layout
- What shall we do?
exec(): high-level outline

- Load program from disk
- Create user-stack
- Run!
int exec(char *path, char **argv)
{
...

  if((ip = namei(path)) == 0) {
    end_op();
    return -1;
  }

  // Check ELF header
  if((readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf)) >
    go to bad;

  if(elf.magic != ELF_MAGIC)
    go to bad;

  goto bad;
int exec(char *path, char **argv)
{
    ...
    if((ip = namei(path)) == 0){
        end_op();
        return -1;
    }
    // Check ELF header
    if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
        goto bad;
    if(elf.magic != ELF_MAGIC)
        goto bad;
    goto bad;
Setup kernel address space()

6333
6334    if((pgdir = setupkvm()) == 0)
6335        goto bad;
6336
Remember: each process maps kernel in its page table
allocuvm(): allocate user pages
loaduvm(): read program from disk
exec(): allocate process' stack

- Allocate two pages
  - One will be stack
  - Mark another one as inaccessible

6361   sz = PGROUNDUP(sz);
6362   if((sz = allocuvm(pgd, sz, sz + 2*PGSIZE)) == 0)
6363       goto bad;
6364   clearpteu(pgd, (char*)(sz - 2*PGSIZE));
6365   sp = sz;
Allocate stack

Virtual

Physical

Page table

Level 1

Level 2
Switch page tables

- Switch page tables
- Deallocate old page table

6398  switchuvm(proc);
6399  freevm(oldpgdir);
6400  return 0;
Wait... which page table we are deallocating?
Wait... which page table we are deallocating?

- Remember `exec()` replaces content of an already existing process
  - That process had a page table
  - We have to deallocate it
Deallocate address space

Virtual

User memory (2GB)

0x80000000 (2GB, KERNBASE)

Kernel memory (2GB)

4GB

Physical

Ununsed by xv6

0xe000000 (PHYSTOP)

234MB

Top of physical memory

Page table

Level 1

0 - 4MB

4 - 8MB

... 

Level 2

2GB - 2GB + 4MB

0 - 4K

4K - 8K

... 

(4MB-4K) - 4MB
Outline: deallocate process address space

- Walk the page table
  - Deallocate all pages mapped by the page table
- Deallocate pages that contain Level 2 of the page-table
- Deallocate page directory
```c
2015 freevm(pde_t *pgdir)
2016 {
2017   uint i;
2018
2019   if(pgdir == 0)
2020     panic("freevm: no pgdir");
2021   deallocuvm(pgdir, KERNBASE, 0);
2022   for(i = 0; i < NPDENTRIES; i++){
2023     if(pgdir[i] & PTE_P){
2024       char * v = P2V(PTE_ADDR(pgdir[i]));
2025       kfree(v);
2026     }
2027   }
2028   kfree((char*)pgdir);
2029 }
```

Deallocate user address space
deallocate(pde_t *pgdir, uint oldsz, uint newsz)
{
... 
    a = PGROUNDUP(newsz);
    for(; a < oldsz; a += PGSIZE){
        pte = walkpgdir(pgdir, (char*)a, 0);
        if(!pte)
            a += (NPTENTRIES - 1) * PGSIZE;
        else if((*pte & PTE_P) != 0){
            pa = PTE_ADDR(*pte);
            if(pa == 0)
                panic("kfree");
            char *v = P2V(pa);
            kfree(v);
            *pte = 0;
        }
    }
    return newsz;
}
1987 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1988 {
...
1995   a = PGROUNDUP(newsz);
1996   for(; a < oldsz; a += PGSIZE){
1997     pte = walkpgdir(pgdir, (char*)a, 0);
1998     if(!pte)
1999       a += (NPTENTRIES − 1) * PGSIZE;
2000     else if((pte & PTE_P) != 0){
2001       pa = PTE_ADDR(*pte);
2002       if(pa == 0)
2003         panic("kfree");
2004       char *v = P2V(pa);
2005       kfree(v);
2006       *pte = 0;
2007     }
2008   }
2009   return newsz;
2010 }
Deallocate Level 2

2015 freevm(pde_t *pgdir)
2016 {
2017   uint i;
2018
2019   if(pgdir == 0)
2020     panic("freevm: no pgdir");
2021   deallocuvm(pgdir, KERNBASE, 0);
2022   for(i = 0; i < NPDENTRIES; i++){
2023     if(pgd[i] & PTE_P){
2024       char * v = P2V(PTE_ADDR(pgd[i]));
2025       kfree(v);
2026     }
2027   }
2028   kfree((char*)pgdir);
Recap

- We know how exec works!
- We can create new processes
Questions?